Продолжаем рассматривать варианты реализаций конструкторов в классах языка C++. И следующим шагом поближе посмотрим на работу конструктора с одним параметром. Для примера я объявлю простой класс, который будет описывать объекты комплексных чисел:

```language-cpp
class Complex {
    double re;
    double im;

public:
    Complex() : re(0.0), im(0.0)
        { }
    Complex(double real) : re(real), im(0.0)
        { }
    Complex(double real, double imag) : re(real), im(imag)
        { }

    void get_data(double& re, double& im)
    {
        re = this->re;
        im = this->im;
    }
};
```

Здесь все вам должно быть знакомо. Очевидно, объекты этого класса допустимо создавать следующими командами:

```language-cpp
int main()
{
    Complex c1;
    Complex c2(0.5);
    Complex c3(-5.4, 7.8);

    return 0;
}
```

Или, вместо круглых скобок, мы можем прописывать фигурные инициализирующие скобки:

```cpp
Complex c1 {};
Complex c2 {0.5};
Complex c3 {-5.4, 7.8};
```

Это полный аналог предыдущего варианта создания объектов класса Complex. Однако мы знаем, что фигурные скобки также можно заменить на обычный знак равенства, который будет аналогом операции инициализации объекта:

```cpp
  Complex c1;
   Complex c2 = 0.5;
   Complex c3 = { -5.4, 7.8 };
```

И, смотрите, что, по сути, происходит в момент создания объекта c2. Справа от инициализатора записано вещественное значение типа double, а слева – формируется объект типа Complex. То есть, из одного типа данных (double) получается объект другого типа данных (Complex) с вызовом конструктора, содержащим один параметр. Получается, что этот конструктор позволяет нам преобразовывать вещественные числа в объекты класса Complex. Именно поэтому такие конструкторы (с одним параметром) получили название **конструкторов преобразования**.

Возможно, на данный момент пока еще не так очевидна их роль. Поэтому я приведу пример реализации метода для прибавления к текущему комплексному числу другого комплексного числа:

```cpp
const Complex& add(const Complex& other)
{
this->re += other.re;
this->im += other.im;
return *this;
}
```

Воспользоваться этим методом можно следующим образом:

```cpp
int main()
{
    Complex c1;
    Complex c2 = 0.5;
    Complex c3 = { -5.4, 7.8 };

    c2.add(c3);
```

```
    double re, im;
    c2.get_data(re, im);

    std::cout << re << " " << im << std::endl;

    return 0;
}
```

Пока все очевидно. Но, смотрите, этот же метод может быть вызван и так:

```
c2.add(4.3);
```

Мы указали обычное вещественное значение, а не объект класса Complex. Тем не менее, все сработало и компилятор «понял», как правильно воспринимать эту команду. Как же ему это удалось? Я, думаю, вы уже догадались. Все дело в конструкторе с одним вещественным параметром. **Компилятор видит, что аргументом метода должен быть объект класса Complex, на входе имеется вещественное число, и он делает попытку создать временный объект типа Complex, инициализируя его этим вещественным значением**. В итоге получается преобразование вещественного числа в тип Complex и метод успешно выполняется.

Видите, какую неожиданную, на первый взгляд, роль играет конструктор с одним параметром, называемый конструктором преобразования.

Ключевое слово explicit

Если по каким-либо причинам неявное преобразование типов следует запретить, то перед конструктором преобразования достаточно прописать ключевое слово explicit, которое было введено в стандарте C++11:

```cpp
explicit Complex(double real) : re(real), im(0.0)
{ }
```

```
В этом случае создание объекта по синтаксису:

```cpp
Complex c2 = 0.5;
```
```

станет невозможным, т.к. предполагает неявный вызов конструктора преобразования. А вот с явным вызовом проблем не возникнет:

```cpp
Complex c2(0.5);
```

или

```cpp
Complex c2{ 0.5 };
```

Аналогично и при вызове метода:

```cpp
c2.add(4.3);  // ошибка (неявный вызов конструктора преобразования)
c2.add(Complex(4.3)); // ok
```

Деструктор класса

Классы языка C++ содержат еще один специальный метод, называемый **деструктор**. Что это такое и как работает я покажу на примере класса представления точек в N-мерном пространстве (с N координатами):

```cpp
class PointND {
    unsigned total {0};
    int *coords {nullptr};
public:
    PointND() : total(0), coords(nullptr)
        { }
    PointND(unsigned sz) : total(sz)
    {
        coords = new int[total] {0};
    }
    PointND(int* cr, unsigned len) : total(len)
    {
        coords = new int[total];
        set_coords(cr, len);
    }

    unsigned get_total() { return total; }
    const int* get_coords() { return coords; }
    void set_coords(int* cr, unsigned len)
    {
        for(unsigned i = 0; i < total; ++i)
            coords[i] = (i < len) ? cr[i] : 0;
    }
};
```

У объектов класса PointND две переменные: total – общее количество координат (размерность пространства); coords – указатель на массив из total целочисленных координат. Изначально эти переменные инициализируются очевидными значениями: 0 и nullptr. Эта же инициализация происходит в конструкторе по умолчанию (без параметров). Либо, можно создать объект с указанной размерностью sz и нулевыми координатами, или с дополнительной передачей массива координат. После конструкторов объявлены простые и очевидные методы.

Некоторые из вас в этом классе сразу заметят, что в конструкторах происходит выделение памяти под массив, но нигде нет ее освобождения. То есть, при создании объектов этого класса возможна утечка памяти. Например:

```language-cpp
int main()
{
    PointND pt(5);  // утечка памяти

    return 0;
}
```

Как это поправить? Для этого в классах предусмотрен еще один специальный метод под названием **деструктор**, обладающий следующими свойствами:

- деструктор вызывается непосредственно перед уничтожением объекта (освобождением памяти, которую он занимает);
- имя метода называется также, как и тип данных с тильдой (‘~’) вначале;
- деструктор ничего не возвращает;
- деструктор не имеет параметров.

Как раз этот метод служит для освобождения всех ресурсов, захваченных текущим объектом. В нашем случае – это выделенная память под массив координат. Поэтому ее освобождение следует прописывать в деструкторе класса PointND:

```cpp
~PointND()
{
delete[] coords;
}
```

Теперь никаких утечек памяти не возникает. Правда, остается ряд проблем, связанных с копированием одного объекта другому. Но эти недостатки мы поправим на следующем занятии.